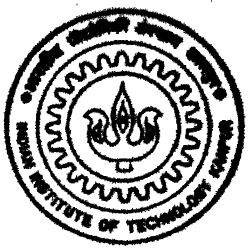
Design and Implementation of an On-lin Software Version Change System for Objective-C

by
KURAVINAKOP SUNIL



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
May, 1997

Design and Implementation of an On-line Software Version Change System for Objective-C

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Kuravinakop Sunil

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

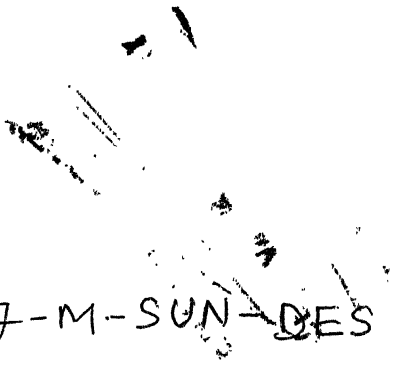
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

May 1997

CERTIFICATE

This is to certify that the work contained in the thesis entitled Design and Implementation of an On-line Software Version Change System for Objective-C by Kuravinakop Sunil has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Deepak Gupta,
Assistant Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology, Kanpur.



Abstract

In this thesis we describe the design and implementation of a prototype for an on-line software version change system for an object-oriented language - Objective-C. The prototypes constructed in the past were either for procedure-based, object-based or for distributed systems. We also describe experiments conducted with the prototype to study its performance. The work reported in this thesis has its foundations in the theoretical work done in the past at IIT Kanpur.

Acknowledgments

I wish to express my sincere gratitude to Dr. Deepak Gupta for his invaluable guidance throughout the work. It was because of his enthusiastic efforts that I could complete this thesis in such a short span. I also thank him for being my mentor and encouraging me in every possible way.

I am thankful to Samir Goel, Samir Shah, Vihari and other friends for going through my report and giving valuable suggestions. I am also thankful to my classmates for making my stay in IIT-K a memorable one.

I am grateful to my parents and my brother for serving as a constant source of motivation.

(Kuravinakop Sunil)

Contents

1 Introduction **1**

1.1 Related Work 1

1.1.1 Dynamic Type Replacement System 2

1.1.2 DAS 2

1.1.3 DYMOS 3

1.1.4 PODUS 4

1.1.5 ARGUS 4

1.1.6 CONIC 5

1.1.7 POLYLITH 6

1.1.8 Past work at IIT Kanpur 6

1.2 Goals and Scope of this work 7

2 Background Concepts **8**

2.1 An On-line Change 8

2.2 Validity of an On-line Change 9

2.3 Determining Validity of an On-line Change 9

2.4 A Brief Overview of Objective-C 10

2.4.1 Features 10

2.4.2 Static Structures 11

2.4.3 Runtime Structures 13

3 Design Issues **16**

3.1 The Problem of Linking 16

3.2 Specification and Implementation of State Mapping 18

3.3 An Example 20
3.4 Specification of the Time of Change 21

4 Implementation **27**
4.1 The modification shell 28
4.1.1 The user interface module 28
4.1.2 The process control module 31
4.1.3 The symbol table management module 31
4.1.4 The run module 31
4.1.5 The replace module 32
4.2 The Modification Library 32
4.2.1 Loader 33
4.2.2 Linker 34
4.2.3 Relocator 34
4.2.4 Objective-C structures updater 35
4.2.5 Information-passer 35
4.2.6 Symbol table management routines 36
4.3 Runtime management module 36
4.4 Initialization routine 38
4.5 Modifications to the GNU C compiler 38

5 Experimentation **39**
5.1 Experiment with the Unix File System Simulator 39
5.2 Experiment with the Memory Simulator Program 40
5.3 Conclusions 41

6 Conclusion **43**
6.1 Limitations 43
6.2 Further Work 44

List of Figures

1 Relevant Static and Runtime Structures of Objective-C 14

2 Objective-C code for first version of the class Queue 22
3 Objective-C code for second version of the class Queue 23
4 Restructuring method 24
5 First version of the class Queue and its Object 24
6 Second version of the class Queue and its Object 25
7 Second version of the class Queue and its Object 25

8 Modules of the Modification Shell 29
9 Modules of the Library 33
10 Version ptrs of classes and objects 37

List of Tables

1 Modification shell commands 30

2 Time taken for on-line change of the file system simulator 40

3 Time taken for on-line change of the memory simulator 41

Chapter 1

Introduction

Although work on programming environments has considerably eased the problems of software development, programmers have neither been able to anticipate all user needs nor have they been able to produce bug free software. Hence computer software needs to be updated quite often. However, not much thought has been given to the problem of changing the software. The usual method of changing the software is to stop the old system and then install the new one. This implies a denial of service to the user during the downtime. This denial of service may be life threatening, for example in an aircraft control system. In a less drastic situation, it may lead to significant monetary losses; consider the case of a stock exchange system for example. Hence there is a need for a change of software version while the current version is executing. We refer to this problem as the on-line software version change problem.

In this thesis we discuss issues relating to the implementation of a prototype using Objective-C as the target language.

1.1 Related Work

Several systems which allow on-line changes to running programs have been implemented in the past. Some work has also been done on the theoretical issues involved in on-line software version change. In this section, we review some of the related work in this area. A good (though slightly dated) survey of the implementations

can be found in [SF93].

Dynamic updation, dynamic modification and dynamic reconfiguration are some of the other terms used in the literature for on-line software version change.

1.1.1 Dynamic Type Replacement System

A dynamic type replacement system was built by Fabry [Fab76]. This is the earliest work reported in this area.

Fabry used a capability based system for performing on-line changes to the implementations of abstract data types. Using a capability based system effectively implies that the address of the procedure to be called is determined at run-time. More precisely, the capability for a procedure points to an instruction that jumps to the procedure. During dynamic updation of a procedure the capability is substituted with a new one using a capability revocation mechanism. The new capability points to the new version of the procedure.

Every data structure contains an additional version number field. Updation of the data structure is not done at the time when dynamic updation is done. When a module uses a data structure for the first time after dynamic updation it checks the version number of the data structure. If this value is older than the current version of the module then the data structure is updated. Since the data structure is updated on demand, it is quite possible for the data structure to lag behind the module by several versions. In this case the previous updation routine is called.

In this system, concurrent accessing and sharing of a data structure among several processes is permitted. Hence every data structure is also provided with a lock apart from version number. Apart from this, synchronization measures between updating processes and ordinary processes have been discussed in detail.

1.1.2 DAS

Goullon et. al. have described the dynamic modification capabilities in the DAS operating system in [HGL78].

Dynamic modification has been implemented through a mechanism called *replugging*. Within a module all the segments (code, data etc.) are described by descriptors which are linked in a linked list. Replugging is performed by changing the links. Each module resides in a separate address space.

A general restructuring scheme has been provided by augmenting every data type with two operations "in" and "out" for "filling" and "emptying" respectively. Instead of using a pairwise restructuring algorithm between different data types, these operations are used. Restructuring of data is done on request rather than on demand.

DAS does not support changes in interface of a function across versions. The system can update all system components including kernel components except the component containing the dynamic modification facility.

A module can be accessed by several processes. Hence synchronization measures between user processes and repluggers have been discussed in detail.

1.1.3 DYMOS

DYnamic MOdification System (DYMOS) [Lee83] was built by Insup Lee for updating programs written in the StarMod language. The unit of change is a procedure. An integrated environment, which includes a command interpreter, an editor, a source code management system, a StarMod compiler and a run-time environment, has been provided. Since the source code, object code and symbol table of a program are all available, they are used while updating the program.

On-line change is done as a series of smaller changes. The set of changed procedures are partitioned into a sequence of pairwise disjoint subsets. In each change a subset of procedures is updated to the new version. Lee gave conditions which these subsets must satisfy in order that the sequence of changes is equivalent to an atomic change.

1.1.4 PODUS

PODUS (Procedure-Oriented Dynamic Updating System) [SF89], was built by Mark Segal and Ophir Frieder. The basic unit of change is a procedure. Each version is in a separate address space with its own binding table. Calls between procedures are always indirect through the binding table.

Dynamic updating is done as a series of incremental changes. Inactive and semantically dependent procedures are updated simultaneously. A procedure is inactive when its invocation is not on the stack and none of the procedures which its new version calls directly or indirectly are on the stack. This also ensures that a procedure in the new version does not call a procedure in the old version. Semantic dependencies between procedures have to be provided by the user, but no guidelines have been provided to determine these. The system also supports updation for next version while updation for previous ones are not over.

Interprocedures are used when the signature of a procedure changes. These are useful when old version procedures have to call procedures in the new version. After a procedure is updated to a new version, in the address space of the older version, the procedure is replaced by an interprocedure.

Mapping procedures, also known as *mprocedures*, are used for restructuring static data in the procedures.

PODUS uses RPC to preserve procedure call semantics across computers. Hence it is suitable for distributed environments.

1.1.5 ARGUS

Argus is a distributed operating system, [Lis88], which provides an integrated environment for developing software in the CLU language. It has facilities for atomic transactions and crash recovery. A program consists of a set of servers called *guardians*. These communicate with each other through an RPC like mechanism.

An updation technique for software in Argus was given by Bloom, [Blo83]. Unit of replacement is a *subsytem* which is a collection of guardians. The condition for acceptable replacement is: the new abstraction should generate only those event

sequences (called *futures*) which would have been permitted for the replaced abstraction in the state in which it is replaced. Based on the above condition, Bloom specifies:

- Relationship between the abstract specifications of the new and old versions.
- Restrictions on the state in which the change can be done.

The above condition implies that the behaviour of a new version should be closely related to that of the old version. Hence sometimes it may be difficult to make those changes which involve bug fixing.

For the updation technique mentioned above, crash recovery facilities of Argus are absolutely essential and hence it may not be adaptable to other systems. Further, for some applications a subsytem may be too large a unit for change.

1.1.6 CONIC

Conic was developed by Kramer and Magee at the Imperial College [KM85]. It provides a language and a run-time environment to construct and reconfigure distributed programs.

In Conic, a program consists of a set of modules. These modules communicate with each other through a set of entry and exit ports. A module does not directly refer to any other module. Instead, a configuration manager creates instances of the modules and sets up interconnection between them. For this, it uses configure specifications (provided by the user) which describe the system configuration.

For updating a system a new set of configuration specifications are provided to the configuration manager. These are used for removing present connections and setting up new connections.

A formal basis to ensure validity of dynamic reconfiguration of distributed programs has been discussed in [KM90]. Exchange of information between processes (nodes) is called a *transaction.* A node that is not currently engaged in a transaction that it initiated and which will not initiate new transactions is called a *passive* node. A node which is passive, not engaged in servicing any transaction and which will

not be engaged in servicing any transactions from other nodes is called as a *quiescent* node. Only a quiescent node can be updated, because it can pass stable and consistent information of its state to the new version. Formal proof for this claim has not been provided. The paper discusses ways by which nodes can be made to achieve this state with a set of passive nodes around them.

1.1.7 POLYLITH

Polylith was developed by Purtilo and Hofmeister in University of Maryland [PH91]. In Polylith, an integrated environment for development and updation of distributed programs for Polylith language has been provided.

Each program consists of a set of modules. A software bus is used to provide support for communication among the modules. Configuration specifications play the same role as in Conic.

In Polylith three types of changes are possible — changes in module implementation, structure of the application and geometry of the application. To change the structure of the application i.e., to add or delete modules and change links between them, no help from the application is required. But to change a module implementation and geometry of the application (i.e., relocate a module to a heterogeneous machine) co-operation of the application is necessary to capture the state of the modules. An *encode* operation in the module is used to capture the state of the module. The encoded state is used by the *decode* operation of the new module to restore state in it.

1.1.8 Past work at IIT Kanpur

An on-line version change system, [GJ93], was implemented on a Sun 3/60 for C programs was earlier developed at IIT Kanpur. A procedure was the unit of change in that system.

The system implemented version change by transferring state of a process running the old software to the state of a process running the new software. In [DGB96], Gupta et. al. considered the theoretical aspects of the version change problem (refer

next chapter for further details). Conditions to ensure validity of an on-line change were specified. Theoretical proofs for these conditions have been given. Using these conditions a set of functions can be derived which should be off the stack during the change in order that the change produces meaningful and acceptable results.

Implementation issues for an object-oriented language have been discussed in [Gup94].

1.2 Goals and Scope of this work

Implementation of an on-line software version change system for object-oriented programs involves several issues which do not arise in other simpler models. To date, there have been implementations for object based models but none for a full-fledged object oriented language.

In this thesis we have attempted to study the issues in the implementation of an on-line software version change system for object-oriented programs. We have built a prototype system for supporting on-line changes to Objective-C programs. We also discuss the experiments conducted to ascertain the performance of the prototype.

The organization of the rest of this thesis is as follows. In chapter 2, we briefly review the theoretical foundation for on-line changes and also the Objective-C language which is the target language for the described implementation. Design issues relating to the implementation of the prototype have been discussed in chapter 3. Implementation of the prototype and experiments conducted with it have been discussed in chapters 4 and 5 respectively. Finally in chapter 6, we conclude this thesis by explaining the limitations of the prototype and giving directions for further work.

Chapter 2

Background Concepts

In [DGB96], a formal framework was developed to understand on-line changes to running programs. In this chapter, we briefly review some of the concepts and ideas developed in that work as a background for the discussion in later chapters. For the sake of completeness, we also give a brief overview of the Objective-C language.

2.1 An On-line Change

The basic idea of an on-line change is to replace the current version of an executing program by a new version of the program.

Gupta et. al. formally defined an on-line change as follows [DGB96].

Definition 2.1 *An on-line change from program old version (Π) to new version (Π') at time t using the state mapping $\mathcal{S}$, in a process P (executing Π) is equivalent to the following sequence of steps:*

1. *P is stopped at time t in state s (say).*
2. *The code of P (which, till now, was the program Π) is replaced by the program Π', its state is mapped by $\mathcal{S}$ and P is then continued (from state $\mathcal{S}(s)$ and with code of Π').*

The state of the process at the time of change may not be suitable for the new version. For example, the new version may have some new variables which did not

exist in the old version at all. Thus it may be necessary to map the state of the process at the time of the change so as to suit the new version of the program. The definition allows this by means of the state mapping $\mathcal{S}$.

2.2 Validity of an On-line Change

It is clear that replacing a program by another program at an arbitrary time need not produce desired and expected results. Therefore we need the notion of *validity* of an on-line change which should capture the user expectations about the behavior of the process after the change. Gupta et. al. [DGB96] defined validity of an on-line change as follows.

Definition 2.2 *An on-line change in the process P from Π to Π' at time t (in state s) and using the state mapping $\mathcal{S}$ is* valid *if after the change, P is guaranteed to reach a reachable state of Π' in a finite amount of time.*

Thus for an on-line change to be valid, the process must reach a reachable (i.e., legal or acceptable) state of the new program version. Once the process reaches this state, its behavior would of course be just as if it had been executing the new program version right from the very beginning.

From the above definition we can see that validity of on-line change is dependent upon the old and new versions of the program, the state mapping function and the time of change. Old and new program versions are provided by the user. The state mapping depends upon the semantic knowledge of the relationship between the program versions and thus must also be supplied by the user. Therefore, to ensure the validity of on-line change, we need to impose conditions on the time of change.

2.3 Determining Validity of an On-line Change

In [DGB96], it was proved that determining validity of an on-line change with given arbitrary parameters (i.e., Π, Π', $\mathcal{S}$, and s) is, in general, undecidable. This implies

that computable necessary and sufficient conditions for determining validity of on-line changes cannot be obtained. This does not, however, preclude computable sufficient conditions for the validity of change.

In [DGB96], Gupta et. al. developed computable sufficient conditions for validity of change for a procedural language model. Procedures were considered as the unit of change and the conditions for validity specified which procedures should be off the stack at the time of change in order to ensure validity. Similar conditions were also given for the object oriented model [Gup94]. Here classes were considered as the units of change and the conditions for validity specified the classes whose methods should not be executing at the time of change. The interested reader is referred to [Gup94] for details.

2.4 A Brief Overview of Objective-C

Objective-C was chosen as the target programming language for this work because, it is a popular object-oriented programming language. Further, its dynamic binding mechanisms and run-time structures make it more amenable to the implementation of on-line software change than C++ [Str90].

In this section, we briefly review the main features of Objective-C and its static and runtime structures (for the GNU compiler). Brief introduction to language syntax can be found at [Dek], [Tig] and [Cra]. Detailed syntax and excellent notes on the runtime system can be found in [Cor].

2.4.1 Features

Objective-C is a super-set of C. The syntax for classes and message passing is similar to that of Smalltalk.

Objective-C supports single inheritance. To implement multiple inheritance, access to the source code of each superclass is essential. This inhibits distribution of code in binary form. Hence multiple inheritance was not implemented. For further details refer to [Cox94].

All classes usually inherit a system defined class called `Object`. This class contains methods which act as an interface between the user and the runtime library. These methods are also useful in handling an object. Some of these are:

- Constructor and destructor methods to allocate and free the object structure, respectively.
- Utility methods to get name of the class, pointer to the class and size of the object.
- Methods for error checking, comparing or making copy of an object etc.

All objects are instantiated at run time. The language also implements true polymorphism. All objects are declared as of type `id`. So, when the compiler parses a method name, it cannot determine the class to which the method belongs and hence cannot determine the address of the method. Hence, method dispatch is dynamic. This dynamism is implemented by using a method dispatch table for each class and before calling a method its address is obtained from the table.

In Objective-C, an object is an instance of a class and the class is considered as an instance of a "metaclass". Classes are also referred to as "class objects". These structures are discussed below.

2.4.2 Static Structures

Structures created at compile time are called static structures. The relevant static structures of Objective-C (for GNU compiler) are:

- Class structure
- Metaclass structure

These structures are shown in figure 1.

Class structure

The class structure stores information about the class, such as the addresses of methods etc. We now discuss the information stored in the class structure in more

detail. Fields `method_list` and `category_list` are pointers to a list of methods in the class. These lists are used to form the *dispatch table*. The `dispatch_table` pointer is used to access the dispatch table. The dispatch table is explained in Section 2.4.3.

All the protocol lists, to which a class conforms, are arranged in a linked list and can be accessed through the `protocol_list` pointer.

The `metaclass_pointer` is used to access the metaclass of the class.

The list of variables and their types, in the class, can be accessed through the `variables` pointer.

All the sibling classes (i.e., subclasses of the same parent class) of the class are arranged in a linked list and can be accessed through the `sibling_class` pointer. The `subclass_list` pointer points to the linked list of subclasses of this class. class list, of the subclasses.

There are other fields like `size` to contain size of the class, `name` to point to the name of the class, `superclass` to point to the superclass, `Version` field to contain the version of the class etc.

Metaclass structure

As mentioned earlier, each class is also considered as an instance of a metaclass. Methods in the metaclass are known as "factory methods" or "class methods". These methods are used to create and initialize objects. In this section we discuss the information stored in the metaclass structure.

Metaclass structure is similar to the class structure. The `size` field is used by the factory methods to allocate an object. The `dispatch_table` and `method_list` fields refer to factory methods, unlike in the class structure where they refer to instance methods. Metaclass structure of the superclass can be obtained through the `superclass` pointer. The `variables`, `size` and `name` fields are identical to those in class structure. Similar to the class structure, the `subclass_list` and `sibling_class` point to the subclass and sibling metaclasses respectively.

2.4.3 Runtime Structures

Structures created during runtime are called as runtime structures. We now discuss the relevant runtime structures of Objective-C (for GNU compiler). The main runtime structures are the following.

- Object structure
- Dispatch table
- Selector table
- Class table

These structures are shown in figure 1.

Object structure

The object structure is allocated during run time using the `size` field present in the Metaclass structure. To create an object, a message is sent to the class object. Using the "factory methods", the object is allocated and initialized.

Object structure contains the instance variables and a pointer to the class structure.

Dispatch table

Each class has a separate dispatch table. The dispatch table contains addresses of methods in a class and the superclass. Before calling a method, its address has to be obtained from the dispatch table.

Each entry in the dispatch table contains address of a method and a pointer to a selector table entry. The class structure contains a method list which is parsed to obtain information for building the dispatch table. Since the method list is implemented as a linked list, the overhead of using the method list during run time is very high and hence the need for a dispatch table.

The dispatch table is implemented as a sparse array.

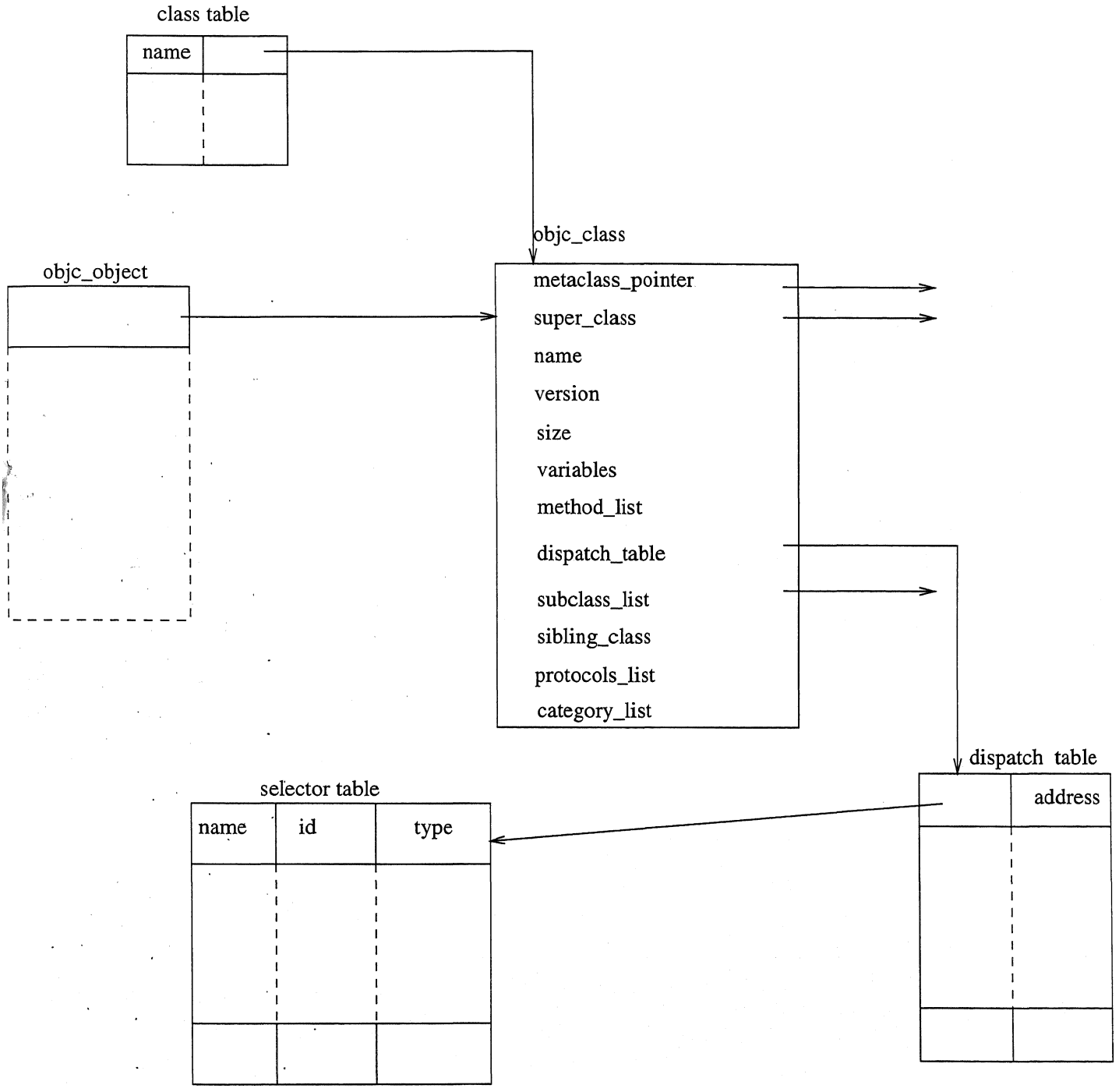


Figure 1: Relevant Static and Runtime Structures of Objective-C

Selector table

Selector table contains method name, runtime identifier and type of the method.

Before using the dispatch table the runtime id of the method has to be obtained using the selector table. This can be done using some functions in the standard Objective-C library. To avoid the overhead of calling these functions each-time a method is called, an array of method ids is formed at runtime which the calling function accesses directly.

There may be several methods of same name and type in different classes. All these are assigned the same runtime id. This helps in optimum usage of space in the dispatch table.

The selector table is implemented as a hash table.

Class table

Class table contains names of all classes and their addresses. When instantiating objects during runtime, address of the class is obtained through the class table.

The class table is implemented as a hash table.

Chapter 3

Design Issues

In this chapter, we briefly describe the high level design of the implemented prototype system for supporting on-line changes to Objective-C programs. We discuss the various design options and our justification for making the choices that we did.

There are two main issues to be resolved in the implementation of an on-line software version change system. The first is, how to link the new code with the already executing, in-memory old code. The second issue is, how to specify and perform the state mapping at the time of change. This issue, as we shall see, is particularly more involved for an object-oriented language.

3.1 The Problem of Linking

This is the main implementation issue in the design of an on-line software version change system. The problem is simply, "how to link the new code with the old already executing code?".

The obvious solution of relinking the whole program at the time of change is clearly too expensive to be feasible. Another solution that has been used previously [Fab76], [SF89] is to make all function calls "indirect" i.e., when the call to a function is made, the starting address of the function is looked up in a table at run-time. For relinking, all one needs to do is to modify this table. However, this solution makes function calls more expensive even when no on-line change is made. Further, it may

require extensive changes to the compiler.

In Gupta's implementation, [GJ93], this problem was avoided altogether by re-linking the new code off-line into a separate executable file. At the time of change, a new process executing the new code is spawned and state is transferred from the old process to the new one. The approach has the merit of being simple but entails a large overhead and has severe limitations unless, the mechanism is implemented in the kernel. Further the approach does not scale well to the object oriented languages, as we shall see in the next section.

Object-oriented languages, like Objective-C and Smalltalk, typically support dynamic binding, i.e., the address of the method called is determined at run-time by table lookup. This is required to support polymorphism, a key feature of object-oriented languages. In this scenario, the problem of relinking at the time of change is simple, one just has to adjust the lookup tables appropriately.

However, Objective-C (as also most other popular object oriented languages) supports both procedural and object-oriented paradigms of programming. Thus, calls to functions are statically linked as in C (i.e., the starting address of the called function is hard coded in the code of the caller function) and object invocations are dynamically linked. Since, we wish to support on-line changes to functions as well as to classes, a mechanism is required to relink calls to functions that have changed in the new version. We decided to achieve this by the simple stratagem of replacing the first instruction of the old version of a function, by a jump instruction to the starting address of the new version. This approach has the merit of being fast, simple, requiring no changes to the compiler and not having the overhead of an indirect call for functions which do not change across versions. The implementation details will be discussed in the next chapter.

When types of global variables change (i.e. new version of the variables are introduced), all the functions and classes accessing this variable are also assumed to have changed. The address of a variable is hard coded, in the binary, in both a function and a method. Since it is not feasible to keep a list of all points at which the variable has been addressed and modify them, the functions and classes accessing the variable have to be re-linked (and therefore specified as having been changed)

even if they have not actually been changed.

3.2 Specification and Implementation of State Mapping

As we saw in the last chapter, the user will typically want to perform a mapping on the state of the process at the time of on-line change. The specification and implementation of such a mapping for a procedural language, such as Pascal, is simple — changing values of some existing global variables and initializing new global variables will suffice in most cases. This can be easily specified by the user by writing a new procedure, an *initialization routine*, which can be called at the time of change as in [GJ93].

In the object oriented case, however, things are more complicated. A change in the code of a class will usually be accompanied by a change in the way the data is stored in the objects of the class. Thus, all objects of the changed classes have to be *restructured* [Gup94]. For example, consider a queue class which uses an array to store the elements of the queue. If the new version of the queue were to use linked lists to store the queue elements, all queue objects which existed at the time of the change would have to be restructured. In other words, linked lists would have to constructed out of the existing arrays. A single user specified function is not sufficient to specify this kind of a state mapping. One needs to allow the user to specify a *restructuring mapping* for each changed class.

Specifying a restructuring mapping for a changed class is not simple either. This is because the restructuring code would have to depend upon both the new and the old versions of the class. According to [Gup94], there are the following three ways in which such a mapping can be specified.

1. The restructuring code uses binary image of the current object. Since, the binary image is dependent on the compiler the restructuring method written would be dependent on the compiler. This is clearly undesirable.

2. The restructuring method uses the instance variables of the old object to extract information from it and uses this information to build the corresponding new one. To differentiate between the current variables and the new variables

of the same names, some syntactical construct has to be designed. In general, this is also not a very attractive alternative as the restructuring code will depend on the actual implementation in the first version of the abstract data type which the class is supposed to implement. Besides, instances variables of one class (i.e., old version) are being directly accessed by methods of another class (i.e., restructuring method of new version), which is opposed to the object-oriented philosophy.

3. Restructuring method calls methods of the old object version to extract information. Clearly for this approach to work the code of both the versions of the class must be available.

The third approach is clearly the most attractive one and has accordingly been chosen for our implementation.

When version change takes place, new restructured objects may be created in lieu of old ones. All other objects and functions referring to the old object must refer to the new one. There are three ways of achieving this:

1. For all objects and functions, any references to the old object are modified. This is possible only if the address space is searched for all references to the old object. This is clearly prohibitive in terms of cost, even if possible.

2. A list of references can be maintained for each object and can be used to update references to the object. But this would incur a possibly unacceptable overhead even in the case where no on-line change is made.

3. A pointer is placed in the old object pointing to the new object. This pointer can be used to "redirect" all calls to the old object to the new version. This approach clearly incurs an overhead only for those objects that have actually been restructured and is most attractive of the three. We therefore chose this approach for our implementation.

Finally the restructuring of objects of changed classes can either be done right at the time of change, or later, "on demand". In the former case, one would again have to keep track of all objects of a given class. Further, the time taken to restructure

all the objects at the time of change may increase the time taken for the change to an unacceptable level. The latter approach has the advantage that objects are restructured only when they are first used after the change. This helps amortize the cost of restructuring. We decided to adopt this approach, therefore, in our design.

To implement restructuring on demand, a new field is added to both the class and the object run-time structures of Objective-C. When a class is changed on-line, a new class structure is created for the new version of the class, and the field `next_version_ptr` in the old class structure is made to point to this new structure. This field is `NULL` for the latest version of the class. Similarly, all objects also carry a field `next_version_ptr`, which points to the next version of the object. However, this field is not changed at the time of change but at the time when actual restructuring occurs. The entry code for all methods checks this field, and the `next_version_ptr` field of the class structure to decide if the object version does not correspond to the latest version of the class. If not, then restructuring is performed and the method code accesses the latest version of the object. In the next section, we present a simple example to further clarify these ideas.

As discussed earlier, types of some variables can change across versions. The values of these variables need to be mapped to the new version. This can be achieved either by distinguishing between the names of variables in the old version and the new version or by letting the programmer obtain the old address of a variable. In our implementation, the old address of a variable can be obtained by using one of the library routines. This is explained in detail in Chapter 4.

3.3 An Example

We now consider a simple example to illustrate how the restructuring mapping is specified and implemented in our implementation. We take the example of a simple queue class which implements a queue of integers using an array. The second version of the class has the same functionality but uses a linked list implementation. The code for the old and the new versions of the class is shown in Figures 2 and 3 respectively.

```
Version 1
#include <object.h>
#define MAX_SIZE 20

/*********************************
* In the first version the
*  circular queue is implemented
*  by an array.
* Maximum size of the queue is
*  MAX_SIZE.
* Array q is used to store the
*  elements.
* Head and tail are indexes into
*  the array and indicate the
*  starting *  and ending of the
*  queue respectively.
*********************************/

// Class Definition
@interface Queue:Object
{
@private //Class variables
      int q[MAX_SIZE];
      int head,tail,size;
}
//Methods of the Class
-init;
-(int)put:(int)id;
-(int)get;
@end
```

```
//Class Implementation
@implementation Queue:Object

-init
{
     head=tail=size=0;
}

// Insert at the tail end.
-(int)put:(int)id
{
     if(!size)
        head=tail=0;
     else if(head==tail) return(-1);
     q[tail]=id;
     size++;
     tail=(tail+1)%MAX_SIZE;
     return(1);
}

// Remove from the head end.
-(int)get
{
     int id;

     if(!size) return(-1);
     size--;
     id=q[head];
     head=(head+1)%MAX_SIZE;
     return(id);
}
@end
```

Figure 2: Objective-C code for first version of the class Queue

```
Version 2

#include <object.h>

/*********************************
* In the second version the queue
*  is implemented by a linked list.
* Structure container is used to
*   implement the linked list.
* Head and tail point to the
*  starting and ending of the queue
*  respectively.
*********************************/

struct container
{
      int id;
      struct container *next;
};

// Interface of the new version of
// the class Queue

@interface Queue:Object
{
@private
   int size;
   struct container *head;
   struct container *tail;
}

-init;
-(int)put:(int)id;
-(int)get;
-mapper:(id)old_object;
@end

@implementation Queue:Object
-init
{
     head=tail=NULL;
     size=0;
     return self;
}

-(int)put:(int)id
{
   if(!size)
   {
       head=tail=
          (struct container *)malloc(sizeof(
                                  struct containe
       head->id=id;
   }
   else
   {
       tail->next=
          (struct container *)malloc(sizeof(
                                  struct containe
       tail=tail->next;
       tail->id=id;
   }
   size++;
   return(1);
}

-(int)get
{
     int contents;
     structure container *o_head;

     o_head=head;
     if(!size) return(-1);

     contents=head->id;
     head=head->next;

     free(o_head);
     size--;
     return(contents);
}

/*****************************************
*Method for mapping (mapper) has to be presen
*here but is shown in a seperate figure, for
*the sake of clarity
*****************************************/
@end
```

Figure 3: Objective-C code for second version of the class Queue

Figure 4 shows the code for restructuring an instance of the old version of the class to a corresponding instance of the new one. The code is specified as a method called mapper of the new version of the queue class.

```
/*************************************
* Mapping method of second version
* of the class Queue. After initializing
* the class, it gets an element from
* the old version of Queue (old_object)
* and puts it in the second version.
* Process is repeated till the old
* queue is totally empty.
*************************************/
-mapper:(id)old_object
{
     int id;

     [self init];
     while((id=[old_object get])!=-1)
          [self put:id];
}
```

Figure 4: Restructuring method

Figure 5 shows the class and the object structures for the queue class and an object o of this class prior to the time the change is initiated.

At the time of on-line change, a new class structure is created for the new version of the class, and the pointers are updated as shown in Figure 6

After the change has been effected, the program resumes execution. Now let us say that a call is made to the put method of the object o. At this time, the method entry code sees that the object version is out of date since the next version pointer of the class of this object is not NULL. It therefore obtains a pointer to the new class structure for the queue class, creates an instance o' of the new queue class and invokes the method mapper on o'. After the method mapper finishes executing, the entry code changes the run-time structures as shown in Figure 7. After this the entry code follows the next version pointer from o to o', verifies that o' is the latest

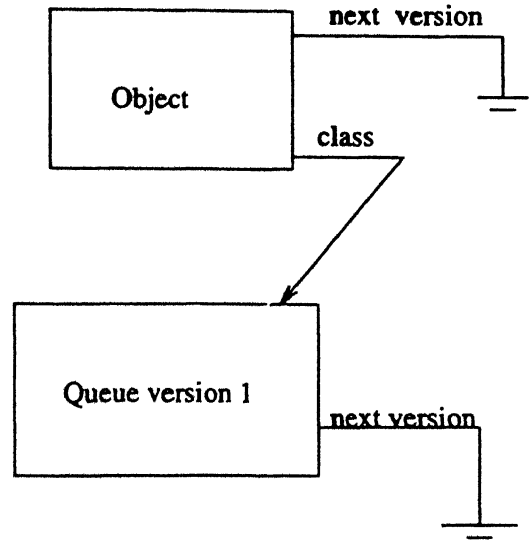


Figure 5: First version of the class Queue and its Object

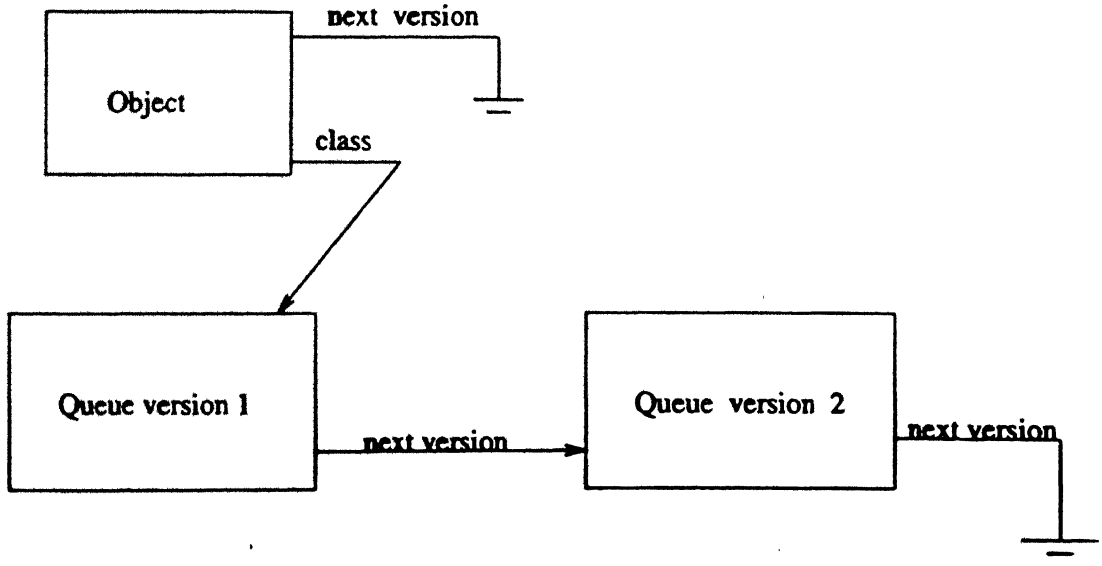


Figure 6: Second version of the class Queue and its Object

version of the object and finally invokes the called method on o'. From now on, all references to o will be correctly forwarded to o' as required.

Note that at the time of restructuring, an object may be several versions out of date, i.e., several version changes may have been effected since the object was last accessed. The process of restructuring allows for this and builds the latest version of the object by invoking the restructuring method of all the versions in succession.

3.4 Specification of the Time of Change

In order that the on-line change is valid, it must be effected at an appropriate time. As stated in the previous chapter, the problem of determining such a time has been shown to be undecidable in general [Gup94]. We therefore require the user to specify the time of change to the system.

In [Gup94], Gupta gave conditions for ensuring validity of change for the procedural and object oriented models of programming. These conditions were in terms of specified procedures and classes respectively being off the stack at the time of change. However, hybrid models (i.e., combining procedural and object-oriented paradigms, as in Objective-C) were not considered. In this work, we have not made any attempt to derive conditions for validity for such a model. However, it seems reasonable to expect the user to specify which functions and classes should be off the stack at the time of change. Thus, similar to the implementation in [GJ93], we require the user to specify the list of functions and classes required to be off the stack at the time of change, as a parameter to the on-line change command.

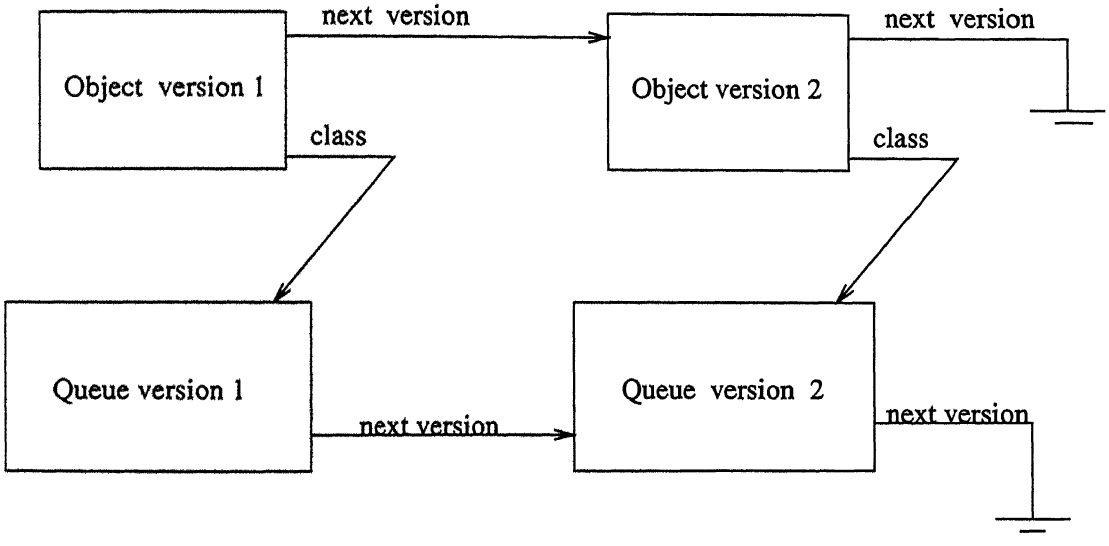


Figure 7: Second version of the class Queue and its Object

Chapter 4

Implementation

The prototype was implemented on a Sun-3 machine running SunOS 4.0. In this chapter, we briefly discuss some of the implementation details of the prototype.

The prototype implementation consists of three main modules:

1. The modification shell.
2. The modification library.
3. The runtime management module.

The modification shell interacts with the user for running and updating the user program etc. The user program is run as a child process of the modification shell.

The modification library provides modules which load, link and relocate the changed parts of the user program. The changed parts of the program are specified as new object code (`.o`) files. The library also provides functions for modifying the Objective-C run-time structures for accomplishing the on-line change. In short, all parts of the process of effecting an on-line change that require changes in the address space of the process executing the user program, are accomplished by the modification library. We require the modification library to be linked with the user program.

The runtime management module refers to the modules that were added to the standard standard Objective-C library.

In the next three sections, we describe these three modules of the implementation in greater detail.

4.1 The modification shell

The modification shell consists of the following modules. The high level design of the shell is illustrated in Figure 8.

1. The user interface module
2. The process control module
3. The symbol table management module
4. The run module
5. The replace module

In the following subsections, we briefly describe the above modules.

4.1.1 The user interface module

The user interface module interacts with the user and accepts commands for running a program, effecting an on-line change to the running program etc. Table 1 shows the complete command set.

The `cd` command changes the working directory of the modification shell. The user process, executing as a child process of the modification shell, can be terminated by the `kill` command. The `tty` command is used to set the terminal for the user process. On using `-e` and `-o` options with "tty", the standard error and standard output terminals, respectively, are set. By default the standard error and standard output terminals are set identical to the standard input terminal. The `run` command starts the execution of the specified program in a child process. On-line version change is performed by the `re` command. This command takes the name of the object file containing the changed code as its first argument. The second argument is the name of the file which contains the list of functions and classes required to

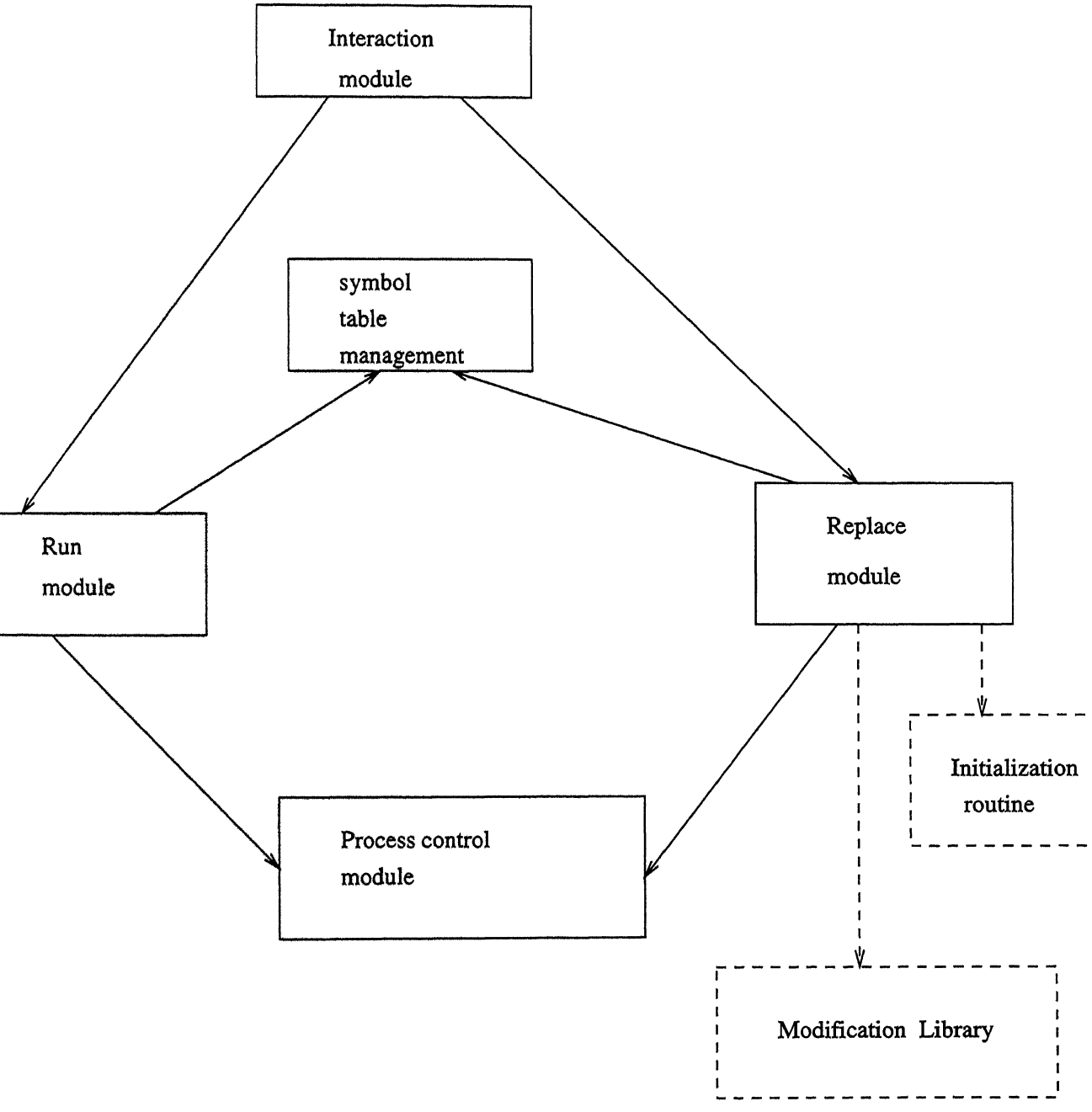


Figure 8: Modules of the Modification Shell

Command	Function
cd dir	Change working directory to "dir".
help [cmd]	Provide online help for command "cmd". If "cmd" is not specified names of all the available commands are displayed.
kill	kill the currently running user process
quit or exit	exit the modification shell, killing user process if any.
re binary off_stack [init]	Replace current version with new version by loading file "binary" into current version, with functions specified in the file "off_stack" to be off the stack at the time of change and with "init" as the initialization routine.
run prog [arg1 ..]	run program "prog".
tty [-e] [-o] [tty-name]	Set I/O terminal to "tty-name". -e option is used to set error terminal -o option is used to set the output terminal By default error and output terminals are same as input terminal
! cmd	Shell escape.

Table 1: Modification shell commands

be off the stack at the time of change. In addition, the name of an initialization routine can be optionally specified. If specified, this routine is executed after the change has been effected. An example of the use of the initialization routine is to initialize any new variables in the program.

4.1.2 The process control module

The process control module contains routines, which aid the replace module, to control the process running the user program (which is spawned as a child process of the modification shell) using the `ptrace` system call [Sun88]. Thus this module provides a simpler and a more convenient interface to the `ptrace` system call for the replace module.

4.1.3 The symbol table management module

The routines in this module aid the replace module in constructing the symbol table of the user program and updating it when version change takes place. The replace module uses this table to check the stack the user process when performing version change as described later.

4.1.4 The run module

The run module is used to start the execution of the specified user program when the user gives the `run` command to the shell. The user program is run as a child process of the modification shell. The steps involved in starting the execution of the specified user program are as follows.

1. Create a child process using `fork`.
2. The child process prepares itself to be traced using the `ptrace` system call, so that the replace module can control it for performing version changes.
3. The child process opens standard input, output and error file streams as specified by the `tty` command.
4. The child process then uses `exec` to start executing the specified user program with the given arguments.

4.1.5 The replace module

The replace module is used to change the version of the current process to the new one.

If the child process is being replaced with new version for the first time then using the symbol table information present in its binary executable file a list of functions in the current version is created. The user specifies the list of functions and classes required to be off the stack at the time of change. The name of the file containing this list is specified as the second parameter of the `re` command. When the `re` command is given, the user process is stopped and its stack is examined to see if any of the specified functions and methods of the classes are on the stack. If none are present, the change is performed immediately. Otherwise the return address from the last such function or method on the stack is modified to the address of an illegal instruction. When the process executes the illegal instruction it receives the SIGILL signal. The parent process i.e., the modification shell waits for this event using the `wait4` system call. The replace module then makes the user process execute functions in the modification library (described later). These functions load and link the specified object code file in the address space of the child process and make the necessary changes to the Objective-C runtime structures. After this, the initialization routine (if specified) is executed in the child process. This completes the on-line change and the child process process then continues its execution from the original point where it was interrupted.

4.2 The Modification Library

The modification library is part of the prototype and has to be linked with the user program during compilation. The library (see figure 9)consists of six modules:

1. Loader
2. Linker
3. Relocater

4. Objective-C structures updater

5. Information-passer

6. Symbol table management routines

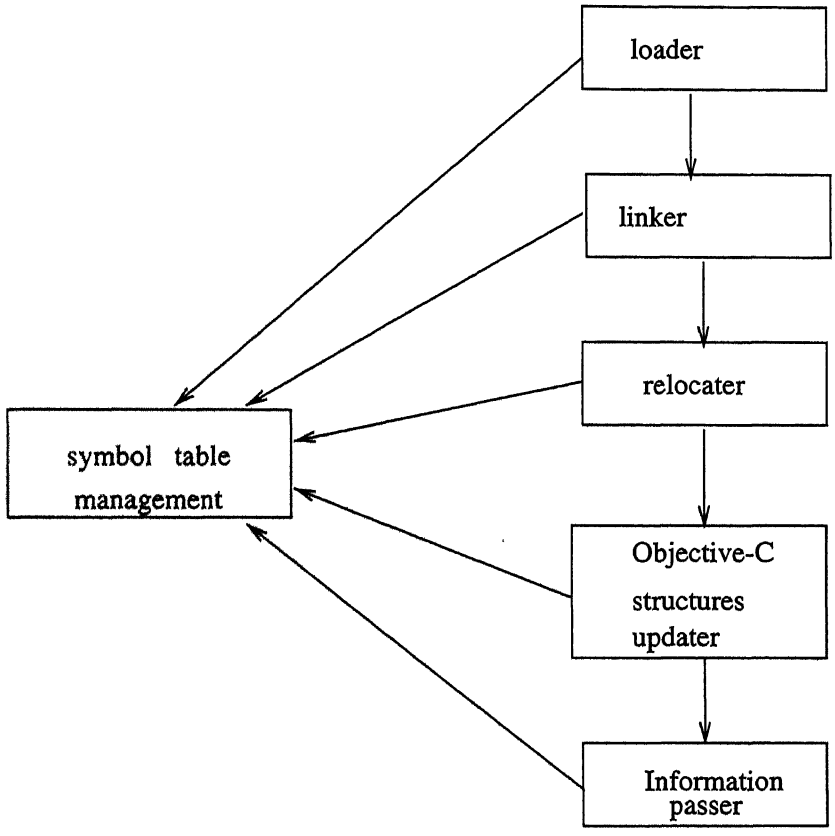


Figure 9: Modules of the Library

These modules operate in the child process and change its address space suitably so as to effect the on-line change. In the following subsections, we briefly discuss the functionality of these modules. We shall, however, not go into the nitty gritty details of linking, relocation etc., for the sake of brevity.

4.2.1 Loader

The loader module builds the global symbol table of the user program using symbol table management routines, when version change is done for first time. It also builds

symbol table from the new object file. It then allocates memory and loads the text segment and initialized data segments of the new object file into the address space of the user process. To load the text segment from new version into text segment of current version, either the loader module should be within the kernel to manipulate size of text segment, or the code should be impure i.e., text and data should be in same segment. Instead of modifying the kernel, a limitation was put on the user program to be impure. Therefore the loader allocates memory and loads the text and initialized data segments of new version into the data area of the user process.

4.2.2 Linker

After the data from the new object file has been loaded in the address space of the user process, the linker is used to resolve the external code and data symbols used in the loaded code and data to existing addresses in the address space of the process. Currently the implementation does not allow the new code to refer to any library not linked in the previous version of the program since this would necessitate loading and linking some part of this library as well. However, this feature can be added easily.

As mentioned in the last chapter, calls to functions that have changed are re-linked by replacing the first instruction (in the old version) by a jump instruction to the new version.

4.2.3 Relocator

The code in the object file is relocatable. This means that the addresses in some of the instructions have to changed after all the symbols have been assigned addresses. This is the job of the relocator module. The relocator module uses the relocation table information in the object file for this purpose. It makes use of the virtual address, symbol number, relocation type and length fields in the relocation table and the addresses at which text and data segments are loaded, provided by the loader, while performing the relocation.

4.2.4 Objective-C structures updater

The next step in effecting the on-line change is to update the runtime Objective-C structures to reflect the change. The Objective-C structures updater module performs this function.

The Objective-C structures updater consists of three sub-modules:

- module to add new classes into the current runtime structures.
- module to update the subclasses.
- module to add new categories.

For adding a new class into the current runtime structures, the class is first added into the class table. If the class table in current version contains a class of the same name then the class is replaced by the new version. The class links i.e., the superclass and subclass links, are resolved. New methods names if any are registered in the selector table and runtime id of all the method names are noted.

The subclasses of the classes inserted above are updated to new versions because,

- the size of the new parent class inserted might have changed. This updation is done recursively till all subclasses are updated.
- new methods might have been added.
- method implementations might have changed.

After insertion and updation, dispatch tables for the new classes are set up using routines in the standard Objective-C run-time library.

4.2.5 Information-passer

The replace module in the modification shell maintains a function table consisting of names of functions in the user program and their addresses. When an on-line change is made, this table has to be updated. The changes made to this information during the on-line change have to be communicated from the modification library (which is

part of the user program) to the replace module (which is part of the modification shell process). The information-passer module arranges these names in an array and passes the total number of functions in the new version to the replace module. The replace module updates its function table by reading this array.

4.2.6 Symbol table management routines

When the version change is done for the first time then, the symbol table information in the binary executable file of the initial version is used to build a symbol table in the library. Whenever version change is done, the linker uses these symbols and their addresses to link with symbols in new version. After the linking is over, the current symbol table is updated with symbols from the new version. The routines in this module manage this task.

4.3 Runtime management module

Two new functions, were added to the Objective-c library:

1. `get_new_version_class`
2. `get_new_version_object`

Given a pointer to a class structure, `get_new_version_class` follows next version pointer to return the latest version of class. For example, in figure 10, given pointer to C_1, `get_new_version_class` follows next version pointer to access C_2. Similarly it accesses C_2, C_3 and C_4. It finally returns a pointer to C_4.

Function `get_new_version_object` accesses the most recent version of the object, (O_2 in figure 10). From this object next version pointer of its class (C_2 in figure 10) is followed. For every new version of the class, new version of the object is created using function `class_create_instance` provided in standard Objective-C runtime library (in figure 10 for classes C_2 and C_3, objects O_3 and O_4 are created). State of the previous version is mapped to the new object using restructuring method (provided in the new version of class) (in figure 10 data from O_2 to O_3 is mapped using restructuring method provided in C_3. Similarly data from O_3 to O_4 is mapped

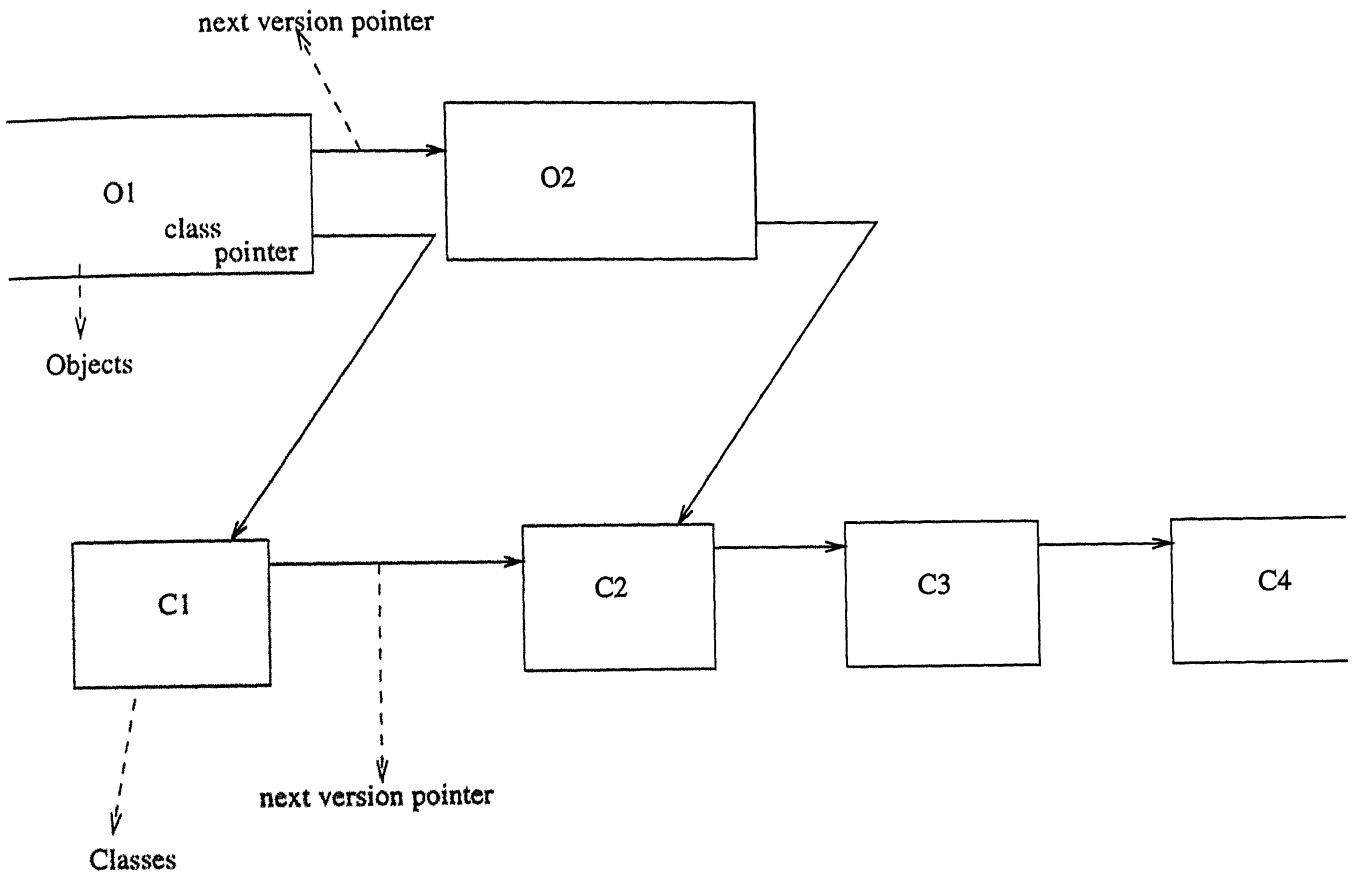


Figure 10: Version ptrs of classes and objects

using restructuring method in C_4). Since restructuring method can map to the new version from its predecessor only, all intermediate versions have to be created (in figure 10 restructuring method in C_4 can map data from O_3 to O_4 only and mapper function in C_3 can map data from O_2 to O_3 only. Therefore O_3 has to created first and data from O_2 to O_3 has to be mapped. Then O_4 has to created and data from O_3 to O_4 has to be mapped.) The most recent version is returned (pointer to O_4 is returned).

Some functions like `objc_msg_lookup`, `class_add_method_list` etc. were modified to call the above two functions. This is done in the beginning of the methods. If the parameter is an object it can be restructured when it is accessed for the first time after the version change of its class.

4.4 Initialization routine

As mentioned earlier in section 4.1.5, the replace module executes an initialization routine in the child process. The initialization routine is specified by the user and is optional and is intended to perform the following functions.

- Initialize new variables introduced in the new version.
- Map values of those variables whose type has changed in the new version. The values can be accessed through the old addresses of the variables, as discussed in the last chapter. The address can be obtained by calling a symbol table management routine *get_old_address*, which returns the address, on providing the name of the old variable.

This routine performs the role of a mapping function discussed in Chapter 2.

4.5 Modifications to the GNU C compiler

Recall that we maintain a pointer to the next version of a class in the class structure. Since the class structure is generated at compile time, the GNU Objective-C compiler was modified to generate an extra field `next_version_ptr` in the class definition.

Chapter 5

Experimentation

Two experiments were conducted in order to study the performance of the prototype. The first experiment used a Unix file system simulator program, while the second used a memory simulator program. In this chapter, we present and discuss the results of the experiments.

5.1 Experiment with the Unix File System Simulator

The Unix file system simulator program used for this experiment simulates a Unix file system within a regular Unix file. It presents an interaction shell to the user where he/she can use familiar Unix commands such as `ls`, `cd`, `mkdir` etc. Files in the simulated file system can be created by copying files from the Unix file system using command `fcp`.

The simulator consists of the following four main classes:

- `Kernel`
- `Inodepool`
- `Bufferpool`
- `DeviceDriver`

change from version :	1 to 2	2 to 3
cpu time taken for change (in msecs)	68	96
real time taken for change (in secs)	12.4	3

Table 2: Time taken for on-line change of the file system simulator

The commands mentioned in the preceding paragraph interact with the kernel through methods like open, `read`, `write`, `mknod`, dup etc. These methods correspond to the same named system calls in Unix.

The inodes are managed by the class `Inodepool`. `Kernel` can interact with `Inodepool` through methods like `iget`, `iput`, `alloc` etc. These methods use secondary classes, `Inode_q` and `I_hash` to maintain a freelist-queue and a hash-table respectively.

Similar to inodes, system buffers are managed by the class `Bufferpool`.

A unix file is used to simulate a secondary device. Data to and from a buffer is read from and written onto this file respectively. This is done by methods in the class `DeviceDriver`.

In version 1, command `ls` gave a simple listing of files in the directory. In version 2 it was enhanced to give a detailed output similar to the one given by the `ls -l` command in Unix. A bug present in the class `Kernel` was rectified while enhancing the simulator from version 2 to version 3.

Table 2 gives the time taken for performing these changes on-line. Five experiments were done and the average of those five were taken.

5.2 Experiment with the Memory Simulator Program

A memory simulator program was used to conduct a second set of experiments. This program simulates page replacement algorithms like First-In First-Out (FIFO) while allocating memory pages to a process. The size of the simulator program is around 1500 lines of code.

A fixed number of frames are used (for memory) in the simulator. For every five

change from version :	1 to 2	2 to 3	3 to 4
cpu time taken for change (in msecs)	28	56	64
real time taken for change (in secs)	7.4	1	1

Table 3: Time taken for on-line change of the memory simulator

pages used by the process, the simulator displays the pages in memory and other statistics. A random number generator is used to simulate the use of pages by the process.

There is only one main class called `Queue`. Methods in the class implement a page replacement algorithm. Also, this class has methods to maintain and display statistics for page usage and pages in memory.

Three version changes were made. In the first version of the simulator, methods in class `Queue` implemented a FIFO algorithm. In the change from version 1 to 2, class `Queue` was updated to a new version. Methods in the new version of the class implemented the Clock Page Replacement algorithm [Tan96]. By updating class Queue to version 3 a bug in the algorithm was corrected. After a change to version 4, methods in class Queue implemented Least Recently Used (LRU) algorithm,

Table 3 gives the time taken for performing these changes on-line. Five experiments were done and the average of those five were taken.

5.3 Conclusions

It can be observed that the cpu time taken for a version change is more than its corresponding time in the preceding version change. This is because sizes of the symbol table and the function table, being maintained by the modification library and the modification shell, increase. Time taken to access these tables forms a major part of the total time taken for a version change.

The response (the real time) taken for the change decreases. This is because while changing from version 1 to 2, version change is being performed for the first time. Hence using the executable file of the first version a symbol table has to be

built by the modification library. Time taken to access a file is not accounted for in the cpu usage time, but is included in the real time. Besides this, the modification shell has to build a function table using the same executable file. There may be other factors involved like swapping of pages etc.

In Gupta's system [Gup94], time taken to perform an on-line change was higher, because it involved transfer of the state of the process running version 1 to the process running version 2. Transfer of state also implies that on-line change is a function of the size of the program. In the system described in this thesis, on-line change is a function of only the change in the program.

Representative values for Guptas's system are given in the table below:

Change from version:	1 to 2	2 to 3	3 to 4	4 to 5
cpu time taken for change (in msecs)	704	776	716	736

Table 4: Time taken for on-line change in Gupta's system

Chapter 6

Conclusion

In this thesis, design issues for building a system for online version change of programs written in Objective-C were discussed. We also described the implementation of the prototype system that we built and experiments conducted with it.

In this chapter we discuss limitations of the prototype and additional work that can be done in the area of on-line software version change.

6.1 Limitations

When version change takes place text segment of the new object file has to be loaded into the text segment of the virtual address space of the running process. For this the loader module (a part of the modification shell) has to be present in the kernel. But in order to avoid modifying the kernel, we put a limitation on the user program to have impure code (i.e., in the executable file text and data segments are in the same segment).

Within a function, a pointer to the object points to the first version. When a message is passed to the object, the `self` pseudo-variable points to the first version of the object in the method. So, at the beginning of the method the chain of version pointers are traversed to get the latest version of the object. For this, the user has to invoke a macro (the `WHILE_MACRO`, defined in file Object.h) at the beginning of the method. The compiler can be modified to generate this code. For lack of time,

we have not however done this.

Variables within an object cannot be accessed directly except by their own methods. If the object has newer versions, then outside the methods the pointer to this object points to the first version. So, when variables are accessed outside these methods, values in the first version get accessed which is undesirable. To overcome this limitation, methods can be declared to return the values of these variables. e.g. `-get_head { return head;}`

The prototype assumes that the user program is sequential. Thus it does not lock any data structures etc. Thus the user program cannot use a threads package for concurrency. However concurrency anyway leads to theoretical problems which have not been satisfactorily solved so far.

6.2 Further Work

Recent years have seen dramatic qualitative shifts in the paradigms of computing. With the explosion of the world wide web, distributed and client-server computing has become omnipresent. Electronic commerce is now a reality; it is now possible, for example, to order airline tickets without leaving one's desk. In such a situation, updating software which provides service to clients all over the world is a real challenge. The main problem here is that one has no control over the clients that may be accessing the service and bringing down a server for even a small period may imply a considerable financial loss. It would therefore be interesting as well as useful to investigate the problem of on-line modification of software providing on-line services on the web.

It would also be useful to extend the present work to distributed object oriented programming. CORBA [Vin97] is a popular framework for distributed object oriented programming and several popular applications are based on this model. It would be interesting to investigate the possibility of implementing on-line version change for CORBA objects. Also real time constraints have not been incorporated into any of the studies done so far in this field. Since real time processes have to produce output at a specific time, on-line change in such environments can be very

useful. On-line change for parallel programs is another direction in which further research can be done.

As suggested in [DG94], ensuring validity of the change (using the formally defined conditions) is difficult to automate. But, tools can be developed to assist the user to accomplish this task.

Bibliography

[Blo83] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, MIT, March 1983.

[BWIDM88] Kernighan Brian W. and Ritchie Dennis M. *"The C programming language"*. Prentice-Hall, Englewood Cliffs, N.J., USA, second edition, 1988. Also available at http://www.netlib.org/pvm3/book/pvm-book.html.

[Cor] NeXT Step Corporation. *"Object-oriented programming and the objective-c language"*. Available by anonymous FTP from *http://www.next.com/Pubs/Documents/OPENSTEP/ObjectiveC/objctoc.htm* Also available at *file:/1d3/www/cseinfo/Info/lang/objc-doc/Next/objctoc.htm* .

[Cox94] Brad J Cox. *"Object Oriented Programming: an Evolutionary Approach"*. Addison-Wesely, 1 edition, 1994.

[Cra] Martin Cracauer. *"A 10 minute introduction to objective-c"*. Available by anonymous FTP from *http://www.geom.umn.edu/software/w3kit/overview/objective-c.html* . Also available at *http://csealpha1:80/Info/lang/objc-doc/10minute/objective-c.html.*

[Dek] Steve Dekorte. *"Objective-C Home Page"*. Available by anonymous FTP from *http://www.batech.com/dekorte/Objective-C/* . Also available at *http://csealpha1:80/Info/lang/objc-doc/home-page/objc.html.*

[DGB96] P. Jalote D. Gupta and G. Barua. "A formal framework for on-line software version change ". *IEEE Transactions on Software Engineering*, 22(2):120–131, Feb 1996.

[Fab76] R. S. Fabry. "How to design a system in which modules can be changed on the fly". *Proceedings 2nd ICSE*, pages 470–476, October 1976.

[FS91] O. Frieder and M. E. Segal. "On dynamically updating a computer program: from concept to prototype". *J. System Software*, 14(2):111–128, September 1991.

[GJ93] Deepak Gupta and Pankaj Jalote. "On-line software version change using state transfer between processes ". *SOFTWARE-PRACTICE AND EXPERIENCE*, 23(9):949–964, September 1993.

[Gup94] Deepak Gupta. *"On-Line software version change"*. Phd thesis, Indian Institute of Technology, Kanpur, November 1994.

[HGL78] Rainer Isle Hannes Goullon and Klans Peter Lohr. "Dynamic restructuring in an experimental operating system". *IEEE transactions on Software Engineering*, SE-4(4):290–307, July 1978.

[KM85] J. Kramer and J. Magee. "Dynamic configuration for distributed systems". *IEEE Trans. on Software Engg.*, SE-11(4):424–436, April 1985.

[KM90] J. Kramer and J. Magee. "The evolving philosophers problem: dynamic change management". *IEEE Trans. on Software Engg.*, 16(11):1293–1306, November 1990.

[Lee83] I. Lee. *DYMOS: A Dynamic Modification System.* PhD thesis, University of Wisconsin, 1983.

[Lis88] B. Liskov. "Distributed programming in Argus". *Communications of the ACM*, pages 300–312, March 1988.

[PH91] J. Purtillo and C. Hofmeister. "Dynamic reconfiguration of distributed programs". In *Proc. IEEE International Conference on Distributed Computing Systems*, pages 560–571, May 1991.

[Pra86] Terrence W. Pratt. *"Programming Languages Design & Implementation"*. Prentice Hall of India, 2 edition, 1986.

[SF89] M. E. Segal and O. Frieder. "Dynamically updating distributed software: supporting change in uncertain and mistrustful environments". In *Proc. IEEE Conference on Software Maintenance*, pages 254–261, October 1989.

[SF93] Mark E Segal and Ophir Frieder. "On the fly program modification: Systems for dynamic updating". *IEEE Software*, pages 53–65, March 1993.

[Str90] Bjarne Stroustrup. *"The design and evolution of C++"*. Addison-Wesley, 2 edition, 1990.

[Sun88] *"Sun OS Reference Manual Release 4.0, System Calls"*. Sun Microsystems, 1988.

[Tan96] Andrew S. Tanenbaum. *"Modern Operating System"*. Prentice Hall of India, 1 edition, 1996.

[Tig] Tiggr. "*Tiggr's Objective-C Page*". Available by anonymous FTP from *http://viper.es.ele.tue.nl/tiggr/objc.html* . Also available at *http://csealpha1:80/Info/lang/objc-doc/objc.html.*

[Vin97] Steve Vinoski. "Corba: integrating diverse applications within distributed heterogeneous environments". *IEEE Communications*, 14(2), Feb 1997.